Vumi Javascript Sandbox Toolkit Documentation

Release 0.1.23

Praekelt Foundation

March 04, 2014

Contents

This is the sandbox toolkit for making writing Javascript applications for Vumi Go's Javascript sandbox.

CHAPTER 1

Interaction Machine

class `ConfigReadEvent` ()

`IMEvent()` fired immediately after sandbox configuration is read.

Arguments

- **im** (*InteractionMachine*) – the interaction machine firing the event.
- **config** (*object*) – the config object.

The event type is `config_read`.

class `IMEvent` ()

An event fired by the interaction machine.

Arguments

- **ev** (*string*) – the event type.
- **im** (*InteractionMachine*) – the interaction machine firing the event.
- **data** (*object*) – additional event data.

class `InboundEventEvent` ()

`IMEvent()` fired when an message status event is received. Typically this is either an acknowledgement or a delivery report for an outbound message that was sent from the sandbox application.

Arguments

- **im** (*InteractionMachine*) – the interaction machine firing the event.
- **event** (*object*) – the event message received.

The event type is `inbound_event`.

class `InteractionMachine` ()

Arguments

- **api** (*SandboxAPI*) – a sandbox API providing access to external resources and inbound messages.
- **state_creator** (*StateCreator*) – a collection of states defining an application.

A controller that handles inbound messages and fires events and handles state transitions in response to those messages. In addition it serves as a bridge between a `StateCreator()` (i.e. set of states defining an application) and resources provided by the sandbox API.

static `api`

A reference to the sandbox API.

static **`api_request`** (*cmd_name*, *cmd*)
Raw request to the sandbox API.

Arguments

- **cmd_name** (*string*) – name of the API request to make.
- **cmd** (*object*) – API request data.

Returns a promise that fires with the response to the API request.

static **`attach`** ()
Register the interaction machine's `on_unknown_command`, `on_inbound_message` and `on_inbound_event` with the sandbox API and set the interaction machine itself as `api.im`.

static **`config`**
The value of the *config* key retrieved from the sandbox config resource. Available once `InteractionMachine.setup_config()` has been called by `InteractionMachine.on_inbound_message()`.

static **`current_state`**
The current `State()` object. Updated whenever a new state is entered via a called to `InteractionMachine.switch_state()`.

static **`done`** ()
Terminate this sandbox instance.

static **`event`** (*event*)
Fire an event from the interaction machine to its state creator.

Arguments

- **event** (*IMEvent*) – the event to fire.

static **`fetch_config_value`** (*key*, *json*, *done*)
Retrieve a value from the sandbox application's Vumi Go config.

Arguments

- **key** (*string*) – name of the configuration item to retrieve.
- **json** (*boolean*) – whether to parse the returned value using `JSON.parse`. Defaults to false.
- **done** (*function*) – function `f(value)` to call once the value has been returned by the config resource.

static **`fetch_translation`** (*lang*, *done*)
Retrieve a `jed` instance containing translations for the given language.

Arguments

- **lang** (*string*) – two letter language code (e.g. `sw`, `en`).
- **done** (*function*) – function `f(jed)` to call with the `jed` instance containing the translations.

Translations are retrieve from the sandbox configuration resource by looking up keys named `translation.<language-code>`.

static **`get_msg`** ()
Returns the inbound user msg object currently being processed by the interaction machine. Returns `null` if no message is being processed.

static `get_user_answer` (*state_name*)
Return the answer stored for a particular state.

Arguments

- **state_name** (*string*) – name of the state to retrieve the answer for.

Returns the value stored or `null` if no value is found.

static `i18n`
A jed gettext object for the current user. Updated whenever a new state is entered via a called to `InteractionMachine.switch_state()`.

static `i18n_lang`
Two-letter language code for the user's language. Updated whenever a new state is entered via a called to `InteractionMachine.switch_state()`.

static `load_user` (*from_addr*)
Load a user's current state from the key-value data store resource.

Arguments

- **from_addr** (*string*) – The address (e.g. MSISN) of the user.

Returns a promise that fires once the user data has been loaded.

static `log` (*message*)
Log a message to the sandbox logging resource at the `info` level.

Arguments

- **message** (*string*) – the log message.

Returns a promise that fires once the log message as been acknowledged by the logging resource.

static `metrics`
A default `MetricStore()` instance. Available once `InteractionMachine.setup_metrics()` has been called by `InteractionMachine.on_inbound_message()`.

static `msg`
The message command currently being processed. Available as soon as `InteractionMachine.on_inbound_message()` is called.

static `on_inbound_event` (*cmd*)
Handle a message event (e.g. an acknowledgement or delivery report).

Arguments

- **cmd** (*object*) – The API request cmd containing the message event.

Fires an `InboundEventEvent()` containing the event.

This method terminates the sandbox once the event has been processed.

static `on_inbound_message` (*cmd*)
Handle an inbound user message triggering state transitions and events as necessary.

Arguments

- **cmd** (*object*) – The API request cmd containing the inbound user message.

The steps performed by this method are roughly:

- `InteractionMachine.setup_config()`
- `InteractionMachine.setup_metrics()`

- `InteractionMachine.load_user()`
- Switch to the user's previous state using `InteractionMachine.switch_state()`.
- Fire a `SessionCloseEvent()`, `SessionNewEvent()` or `SessionResumeEvent()` event as appropriate.
- Call the current state's `input_event` method for resumed sessions of the current state's `new_session_event` method for new sessions.
- Send a reply from the current state if the session was not closed.

Afterwards this method terminates the sandbox.

static `on_unknown_command`(*cmd*)

Called by the sandbox API when a command without a handler is received. Logs an error and terminates the sandbox instance.

Arguments

- **cmd** (*object*) – The API request that no command handler was found for.

The handlers currently implemented are:

- `InteractionMachine.on_inbound_message()`
- `InteractionMachine.on_inbound_event()`

static `refresh_i18n`()

Re-fetches the appropriate language translations if the user's language setting has changed since translations were last loaded. Sets `self.i18n` to a new `jed` instance and sets `self.i18n_lang` to `self.user.lang`.

Returns a promise that fires once the translations have been refreshed.

static `reply`(*msg*, *save_user*)

Send a response from the current state to the user.

Arguments

- **msg** (*object*) – the inbound message being replied to.
- **save_user** (*boolean*) – whether to save the user state.

Returns a promise which fires once the response has been sent and the user state successfully stored.

static `set_user_answer`(*state_name*, *answer*)

Sets the answer for the given state.

Arguments

- **state_name** (*string*) – name of the state the answer is for.
- **value** (*object*) – value of the answer (usually a string).

This is called by `State()` objects when they determine that the user has submitted an answer.

static `set_user_lang`(*lang*)

Arguments

- **lang** (*string*) – The two-letter code of the language the user has selected. E.g. *en*, *sw*.

static `set_user_state`(*state_name*)

Sets the stored value of the user's current state.

Arguments

- **state_name** (*string*) – name of the state the user is now in.

This only sets the stored value of the user's state. Actual state changes are handled by `switch_state()` which calls this method.

static `setup_config`()

Retrieves the sandbox config and stores it on the interaction machine as `self.config` for later use. Fires a `ConfigReadEvent()` so that state creators may perform application specific setup.

static `setup_metrics`()

Assign a `MetricStore()` instance to `self.metrics`. The store name is read from `self.config.metric_store` (with the name `default` as the default).

static `state_creator`

A reference to the `StateCreator()`.

static `store_user`(*from_addr*, *user*)

Save a user's current state to the key-value data store resource.

Arguments

- **from_addr** (*string*) – The address (e.g. MSISN) of the user.
- **user** (*object*) – The user state to save.

Returns a promise that fires once the user data has been saved.

static `switch_state`(*state_name*)

Switch to a new state.

Arguments

- **state_name** (*string*) – Name of the state to switch to.

This method returns a promise that fires once the state transition is complete.

If the current state has the same name as `state_name`, no state transition occurs. Fires `StateExitEvent()` and `StateEnterEvent()` events as appropriate.

static `user`

User data for the current user. Available once `InteractionMachine.load_user()` has been called by `InteractionMachine.on_inbound_message()`.

static `user_addr`

Address of current user (e.g. their MSISDN). Available once `InteractionMachine.load_user()` has been called by `InteractionMachine.on_inbound_message()`.

static `user_key`(*from_addr*)

Return the key under which to store user state for the given `from_addr`.

Arguments

- **from_addr** (*string*) – The address (e.g. MSISDN) of the user.

User state may be namespaced by setting `config.user_store` to a prefix to store the application's users under.

class `NewUserEvent`()

`IMEvent()` fired when a message arrives from a user for whom there is no user state (i.e. a new unique user).

Arguments

- **im** (*InteractionMachine*) – the interaction machine firing the event.

The event type is `new_user`.

class **`SessionCloseEvent`**()

`IMEvent()` fired when a user session ends.

Arguments

- **im** (*InteractionMachine*) – the interaction machine firing the event.
- **boolean** (*possible_timeout*) – true if the session was terminated by the user (including when the user session times out) and false if the session was closed explicitly by the sandbox application.

The event type is `session_close`.

class **`SessionNewEvent`**()

`IMEvent()` fired when a new user session starts.

Arguments

- **im** (*InteractionMachine*) – the interaction machine firing the event.

The event type is `session_new`.

class **`SessionResumeEvent`**()

`IMEvent()` fired when a new user message arrives for an existing user session.

Arguments

- **im** (*InteractionMachine*) – the interaction machine firing the event.

The event type is `session_resume`.

`State.`**`start_state_creator`**(*state_name*, *im*)

Arguments

- **state_name** (*string*) – the name of the start state.
- **im** (*InteractionMachine*) – the interaction machine the start state is for.

This default implemenation looks up a creator for the state named `state_name` and calls that. If no such creator exists, it calls `error_state_creator` instead.

class **`StateCreator`**()

Arguments

- **start_state** (*string*) – Name of the initial state. New users will enter this state when they first interact with the sandbox application.

A set of states defining a sandbox application. States may be either statically created via `add_state`, dynamically loaded via `add_creator` or completely dynamically defined by overriding `switch_state`.

static **`add_creator`**(*state_name*, *state_creation_function*)

Arguments

- **state_name** (*string*) – name of the state
- **state_creation_function** (*function*) – A function `func(state_name, im)` for creating the state. This function should take the state name and interaction machine as parameters and should return a state object either directly or via a promise.

static **`add_state`**(*state*, *translate*)

Arguments

- **state** (*State*) – the state to add.

- **translate** (*boolean*) – whether the state should be re-translated each time it is accessed. The default is true.

static **`error_state_creator`** (*state_name*, *im*)

Arguments

- **state_name** (*string*) – the name of the state for which an error occurred.
- **im** (*InteractionMachine*) – the interaction machine in which the error occurred.

This default implementation creates an EndState with name `state_name` and content *"An error occurred. Please try again later"*.

The end state created has the next state set to null so that:

- It won't set the next state.
- When `switch_state()` is next reached, we identify that the user currently has no state, and use the start state instead.

If the start state still does not exist, another error state will be created.

static **`on_event`** (*event*)

Arguments

- **event** (*IMEvent*) – the event being fired.

Called by the interaction machine when an `IMEvent()` is fired. This method dispatches events to handler methods named `on_<event_type>` if such a handler exist.

Handlers should accept a single parameter, namely the event being fired.

Handler methods may return promises.

static **`switch_state`** (*state_name*, *im*)

Arguments

- **state_name** (*string*) – the name of the state to switch to.
- **im** (*InteractionMachine*) – the interaction machine the state is for.

Looks up a creator for the given state_name and calls it. If the state name is undefined, calls `start_state_creator` instead.

This function returns a promise.

It may be overridden by `StateCreator()` subclasses that wish to provide a completely dynamic set of states.

class **`StateEnterEvent`** ()

`IMEvent()` fired when a user enters a state from a different state.

Arguments

- **im** (*InteractionMachine*) – the interaction machine firing the event.
- **state** (*object*) – the state object being entered.

The event type is `state_enter`.

class **`StateExitEvent`** ()

`IMEvent()` fired when a user exits a state to a different state.

Arguments

- **im** (*InteractionMachine*) – the interaction machine firing the event.

- **state** (*object*) – the state being left.
- **user** (*object*) – the user leaving the state.

The event type is `state_exit`.

CHAPTER 2

States

class **`State`**(*name*[, *handlers*])

Base class for application states.

Arguments

- **name** (*string*) – name of the state.
- **handlers** (*object*) – Mapping of handler names to handler functions for state events. Possible handler names are `setup_state`, `on_enter` and `on_exit`. Handler functions have the form `func(state)`.

class **`BookletState`**(*name*[, *opts*])

A state for displaying paginated text.

Arguments

- **name** (*string*) – name of the state
- **opts.next** (*fn_or_str*) – state that the user should visit after this state. Functions should have the form `f(message_content [, done])` and return the name of the next state. If the `done` argument is present, `f` should arrange for the name of the next state to be return asynchronously using a call to `done(state_name)`. The value of `this` inside `f` will be the calling `BookletState()` instance.
- **opts.pages** (*integer*) – total number of pages.
- **opts.page_text** (*function*) – function `func(n)` returning the text of page `n`. Pages are numbered from 0 to (pages - 1). May return a promise.
- **opts.initial_page** (*integer*) – page number to use when the state is entered. Optional, default is 0.
- **opts.buttons** (*object*) – map of user inputs to amounts to increment the page number by. The special value 'exit' triggers moving to the next state. Optional, default is: `{"1": -1, "2": +1, "0": "exit"}`,
- **opts.footer_text** (*string*) – text to append to every page. Optional, default is: `"\n1 for prev, 2 for next, 0 to end."`
- **opts.handlers** (*object*) – object of handlers for particular events, see `State()`.

class **`Choice`**(*value*, *label*)

An individual choice that the user can select inside a `ChoiceState()`.

Arguments

- **value** (*string*) – string used when storing, processing and looking up the choice.

- **label** (*string*) – string displayed to the user.

class **`ChoiceState`**(*name*, *next*, *question*, *choices*, *error*, *handlers*[, *options*])

A state which displays a list of numbered choices, then allows the user to respond by selecting one of the choices.

Arguments

- **name** (*string*) – name used to identify and refer to the state
- **next** (*fn_or_str*) – state that the user should visit after this state. Functions should have the form `f(choice [, done])` and return the name of the next state. If the `done` argument is present, `f` should arrange for the name of the next state to be return asynchronously using a call to `done(state_name)`. The value of `this` inside `f` will be the calling `ChoiceState()` instance.
- **question** (*string*) – text to display to the user
- **error** (*string*) – error text to display to the user if we reach this state in error. Optional.
- **handlers** (*object*) – object of handlers for particular events, see `State()`.
- **options.accept_labels** (*boolean*) – whether choice labels are accepted as the user's responses. For eg, if `accept_labels` is true, the state will accepts both "1" and "Red" as responses responses if the state's first choice is "Red". Defaults to `false`.

class **`LanguageChoice`**(*name*, *next*, *question*, *choices*, *error*, *handlers*[, *options*])

A state for selecting a language.

It functions exactly like `ChoiceState()` except that it also stores the value of the selected choices as the user's language (it is still available as an answer too).

`Choice()` instances passed to this state should have two-letter language codes as values, e.g.:

```
new LanguageChoice(
    "select_language",
    "next_state",
    "What language would you like to use?",
    [ new Choice("sw", "Swahili"), new Choice("en", "English") ]
);
```

See `ChoiceState()` for a description of the parameters to the constructor.

class **`PaginatedChoiceState`**(*name*, *next*, *question*, *choices*, *error*, *handlers*, *page_opts*[, *options*])

A sub-class of `ChoiceState()` that splits the list of choices given into pages.

Arguments

- **page_opts.back** (*string*) – Label to use for the previous page option (defaults to `Back`).
- **page_opts.more** (*string*) – Label to use for the next page options (defaults to `More`).
- **page_opts.options_per_page** (*integer*) – Maximum number of options per page, excluding the next and previous page options (defaults to 8).
- **page_opts.characters_per_page** (*interger*) – If set, labels for choices will be shortened so that there are no more than this number of characters per page of text sent to the user (default is `null` – i.e. don't shorten any text). Shortened choices are truncated and have `...` appended to indicate to the user that the option has been shortened. The character count includes all content rendered when displaying the state (i.e. the question, the choices selected for the page and any previous or next choices added by `PaginatedChoiceState()`).

Other parameters are described in `ChoiceState()`.

class **`EndState`** (*name*, *text*, *next*, *handlers*)

A state which displays a list of numbered choices, then allows the user to respond by selecting one of the choices.

Arguments

- **name** (*string*) – name used to identify and refer to the state
- **text** (*string*) – text to display to the user
- **next** (*fn_or_str*) – state that the user should visit after this state. Functions should have the form `f(message_content)` and return the name of the next state. The value of `this` will be the calling `EndState()` instance. If `next` is `null`, the state machine will be left in the current state.
- **handlers** (*object*) – object of handlers for particular events, see `State()`.

class **`FreeText`** (*name*, *next*, *question*, *check*, *error*, *handlers*)

A state which displays a text, then allows the user to respond with any text.

Arguments

- **name** (*string*) – name used to identify and refer to the state
- **next** (*fn_or_str*) – state that the user should visit after this state. Functions should have the form `f(message_content [, done])` and return the name of the next state. If the `done` argument is present, `f` should arrange for the name of the next state to be return asynchronously using a call to `done(state_name)`. The value of `this` inside `f` will be the calling `FreeText()` instance.
- **question** (*string*) – text to display to the user
- **check** (*function*) – a function `func(content)` for validating a user's response. Should return `true` if the response is considered valid, and `false` if otherwise.
- **error** (*string*) – error text to display to the user if we reach this state in error. Optional.
- **handlers** (*object*) – object of handlers for particular events, see `State()`.

CHAPTER 3

HTTP API

class **`HttpApi`**(*im*, *opts*)

A helper class for making HTTP requests via the HTTP sandbox resource.

Arguments

- **im** (*InteractionMachine*) – The interaction machine to use when making requests.
- **opts.headers** (*object*) – Default headers to use in HTTP requests.
- **opts.auth** (*object*) – Adds a HTTP Basic authentication to the default headers. Should contain `username` and `password` attributes.

static **`check_reply`**(*reply*, *method*, *url*, *request_body*)

Check an HTTP reply and raise an `HttpApiError()` if the response status code is not in the 200 range or the reply body cannot be decoded.

Logs an error via the sandbox logging resource in an error is raised.

Arguments

- **reply** (*object*) – Raw response to the sandbox API command.
- **method** (*string*) – The HTTP method used in the request (for use in error messages).
- **url** (*string*) – The URL the HTTP request was made to (for use in error messages).
- **request_body** (*string*) – The body of the HTTP request (for use in error messages).

Returns the decoded response body or raises an `HttpApiError()`.

static **`create_headers`**(*headers*)

Combines a set of custom headers with the default headers passed to `HttpApi()`.

Arguments

- **headers** (*object*) – Additional HTTP headers. Attributes are header names. Values are header values.

Returns the complete set of HTTP headers.

static **`decode_response_body`**(*body*)

Arguments

- **body** (*string*) – The body to decode.

Sub-classes should override this to decode the response body and throw an exception if the body cannot be parsed. This base implementation returns the body as-is (i.e. decoding is left to the code calling the `HttpApi()`).

static `delete` (*url*, *opts*)
Make an HTTP DELETE request.

Arguments

- **url** (*string*) – The URL to make the request to.
- **opts** (*object*) – Options to pass to `HttpApi.request()`.

Returns a promise which fires with the decoded value of the response body or an object with an error attribute containing the error message.

static `encode_request_data` (*data*)

Arguments

- **data** (*object*) – The data to encode.

Sub-classes should override this to encode the request body and throw an exception if the data cannot be encoded. This base implementation returns the data as-is (i.e. encoding is left to code calling the `HttpApi()`).

static `get` (*url*, *opts*)
Make an HTTP GET request.

Arguments

- **url** (*string*) – The URL to make the request to.
- **opts** (*object*) – Options to pass to `HttpApi.request()`.

Returns a promise which fires with the decoded value of the response body or an object with an error attribute containing the error message.

static `head` (*url*, *opts*)
Make an HTTP HEAD request.

Arguments

- **url** (*string*) – The URL to make the request to.
- **opts** (*object*) – Options to pass to `HttpApi.request()`.

Returns a promise which fires with the decoded value of the response body or an object with an error attribute containing the error message.

static `post` (*url*, *opts*)
Make an HTTP POST request.

Arguments

- **url** (*string*) – The URL to make the request to.
- **opts** (*object*) – Options to pass to `HttpApi.request()`.

Returns a promise which fires with the decoded value of the response body or an object with an error attribute containing the error message.

static `put` (*url*, *opts*)
Make an HTTP PUT request.

Arguments

- **url** (*string*) – The URL to make the request to.
- **opts** (*object*) – Options to pass to `HttpApi.request()`.

Returns a promise which fires with the decoded value of the response body or an object with an error attribute containing the error message.

static `request` (*method*, *url*, *opts*)

Arguments

- **method** (*string*) – The HTTP method to use (e.g. *GET*, *POST*).
- **url** (*string*) – The URL to make the request to. If you pass in query parameters using `opts.params`, don't include any in the URL itself.
- **opts.params** (*object*) – Key-value pairs to append to the URL as query parameters.
- **opts.data** (*object*) – Data to pass as the request body. Will be encoded using `HttpApi.encode_request_data()` before being sent.
- **opts.headers** (*object*) – Additional headers to add to the default headers.

Returns a promise that fires once the request is completed. The value returned by the promise is the body of the response decoded using `HttpApi.decode_response_body()` or an object containing an attribute named *error* whose value is the error message.

class `HttpApiError` (*msg*)

Thrown when an error occurs while making an HTTP request.

Arguments

- **msg** (*string*) – a description of the error.

class `JsonApi` (*im*, *opts*)

A helper class for making HTTP requests that send and receive JSON encoded data.

Arguments

- **im** (*InteractionMachine*) – The interaction machine to use when making requests.
- **opts.headers** (*object*) – Default headers to use in HTTP requests. The `Content-Type` header is overridden to be `application/json; charset=utf-8`.
- **opts.auth** (*object*) – Adds a HTTP Basic authentication to the default headers. Should contain `username` and `password` attributes.

static `decode_response_body` (*body*)

Decode an HTTP response body using `JSON.parse()`.

Arguments

- **body** (*string*) – Raw HTTP response body to parse.

Returns the decoded response body.

static `encode_request_data` (*data*)

Encode an object as JSON using `JSON.stringify()`.

Arguments

- **data** (*object*) – Object to encode to JSON.

Returns the serialized object as a string.

See also Vumi Go's documentation.

CHAPTER 4

Example Applications

To get you started, here are some example applications that may be useful as an example or reference.

4.1 JSBox Skeleton

A bare bones application that you can use as a starting point. It's ready for you to read, adapt, unit-test, deploy and use on your phone within minutes.

4.2 Contacts Example

You can create, update and remove contact information in Vumi Go's contact database. Here is an example application that shows you how.

4.3 Groups Example

Want to access Vumi Go's groups? The Go Groups application shows you how to do that. It's a simple application that lets you create, list, and search for groups via USSD.

4.4 Key Value Store Example

Want to store some data for your application? Have a look at the Key Value store example application. Useful for if you need to maintain counters across sessions or have some session information you want to hold on to.

4.5 Booklets!

Sometimes you have little nuggest of information that's shareable via USSD. Specifically for that we've created the Booklet State. It allows you to page through information over USSD. Here is an example application that uses it.

4.6 SMS keywords

An often used pattern with SMS shortcodes is to assign different behaviour to different keywords. The sms keyword application shows you how that can be done.

4.7 Events & Metrics

Want to track growth or changes in your application over time? The events firing example application shows you how that can be done. In the background this publishes events to Graphite.

4.8 Google Maps Mashup

An example mashup combining USSD, Google Map's APIs and SMS. See how all these fit together to create a super useful application that does geolocation and delivery of directions via USSD & SMS.

Note: This application is available in South Africa on `*120*8864*1105#`.

4.9 Ushahidi

We're big fans of Ushahidi, the crisis mapping tool. This Ushahidi USSD app is another mashup of USSD and the Ushahidi API. Allows reporting of geolocated events via USSD to hosted Ushahidi instances.

Note: This application is available in South Africa on `*120*8864*1087#`.

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*